

## Article

# SJBCD: A Java Code Clone Detection Method Based on Bytecode Using Siamese Neural Network

Bangrui Wan <sup>1,2</sup>, Shuang Dong <sup>1</sup>, Jianjun Zhou <sup>1</sup> and Ying Qian <sup>1,2,\*</sup><sup>1</sup> School of Software Engineering, Chongqing University of Posts and Telecommunications, Chongqing 400065, China<sup>2</sup> Chongqing Engineering Research Center of Software Quality Assurance, Testing and Assessment, Chongqing 400065, China

\* Correspondence: qianying@cqupt.edu.cn

**Abstract:** Code clone detection is an important research topic in the field of software engineering. It is significant in developing software and solving software infringement disputes to discover code clone phenomenon effectively in and between software systems. In practical engineering applications, clone detection can usually only be performed on the compiled code due to the unavailability of the source code. Additionally, there is room for improvement in the detection effect of existing methods based on bytecode. Based on the above reasons, this paper proposes a novel code clone detection method for Java bytecode: SJBCD. SJBCD extracts opcode sequences from byte code files, use GloVe to vectorize opcodes, and builds a Siamese neural network based on GRU to perform supervised training. Then the trained network is used to detect code clones. In order to prove the effectiveness of SJBCD, this paper conducts validation experiments using the BigCloneBench dataset and provides a comparative analysis with four other methods. Experimental results show the effectiveness of the SJBCD method.

**Keywords:** code clone; code clone detection; bytecode; siamese neural network



**Citation:** Wan, B.; Dong, S.; Zhou, J.; Qian, Y. SJBCD: A Java Code Clone Detection Method Based on Bytecode Using Siamese Neural Network. *Appl. Sci.* **2023**, *13*, 9580. <https://doi.org/10.3390/app13179580>

Academic Editor: Christos Bouras

Received: 3 July 2023

Revised: 3 August 2023

Accepted: 22 August 2023

Published: 24 August 2023



**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Code clones are multiple identical or similar code fragments that exist in a software system, code repository, etc. Although code clones bring convenience to software development, they affect the iteration and maintenance of software systems negatively [1,2]. Code clones may lead to continuous expansion of software systems and increase maintenance costs [3]. In addition, code clones also provide help for the propagation of potential software defects, reducing the reliability of software systems. It is significant to research code clone detection for software quality assurance [4]. Furthermore, code clone detection technology can also be used to analyze software copyright infringement issues. In practical engineering applications, source code is usually unavailable. Only compiled code can be obtained. Beyond that, software programs are often obfuscated by obfuscation tools for security reasons, so it is difficult to reverse them into source code. In this case, it is particularly important to be able to perform code clone detection based on compiled code.

Bellon et al. [5] classified code clones into three types: identical code fragments (Type-1), renamed code fragments (Type-2) and nearly identical code fragments (Type-3). Svajlenko et al. [6] added a new clone definition: syntactically dissimilar code fragments that implement the same functionality (Type-4), made a more detailed division of Type-3 and Type-4, and determined clone categories based on text similarity. The degree is divided into the following types: Very-Strongly Type-3 clones (VST3), Strongly Type-3 (ST3), Moderately Type-3 (MT3), and Weakly Type-3/Type-4 (WT3/T4). The text similarities of four categories are [90%, 100%), [70%, 90%), [50%, 70%), and [0, 50%), respectively. The similarity is judged by diff. Diff is a tool to compare textual differences. For convenience

of description, T1, T2, T3 and T4 are used below to represent Type-1, Type-2, Type-3, and Type-4 clones, respectively.

The detection difficulty of the T1 to T4 clones gradually increases. At present, many scholars convert the code into syntax trees, program flow graphs, etc., and apply deep learning technologies. These methods have great effects on detecting the most difficult WT3/T4. However, most of the methods serve source code. In some scenarios, source code is not easy to obtain, and only compiled code can be obtained. For example, in the scenarios of commercial software plagiarism detection, compiled code difference detection, etc., only compiled code can be obtained. At this time, the detection method based on bytecode can play an effective role. At present, there are limited methods available for bytecode detection, and SeByte [7] provides a metrics-based approach. In SeByte's paper, the recall rate for small datasets was reported to be only 94%. Yu et al. [8] have the capability to detect code clones at both the method level and block level by utilizing block-level code fragments extracted from the bytecode. However, their method can only detect clone pairs of Type-3 at most. Subsequently, Yu et al. [9] proposed a method based on bytecode sequence alignment, which slightly improved the detection effectiveness compared to their previous work. However, it is important to note that these methods were proposed earlier and have not been closely integrated with deep learning. It can be predicted that there is still significant room for improvement in the detection effectiveness. Therefore, there is an urgent need for a code clone detection method that can detect code clones both at bytecode and ensure the detection effect.

### 1.1. Terminologies

Bytecode is compiled from source code. This chapter combines the characteristics of bytecode to analyze its advantages in code clone detection.

In order to ensure the accuracy of description, this paper explains terminologies related to bytecode as follows:

1. Bytecode: Bytecode is only available in specific programming languages, such as Java, Scala, Groovy, and Kotlin. It is compiled from source code and is binary code for Java virtual machine. Unless otherwise specified, the bytecode referred to in this paper are all Java bytecode (codes compiled from Java programs).
2. Opcode: An opcode is a number that represents an operation of the program on the Java virtual machine. For ease of understanding, Oracle introduced the corresponding mnemonic. For example, the mnemonic corresponding to "0x01" is "aconst\_null", which means that null is loaded to the top of the operand stack. Unless otherwise specified, the opcodes referred to in this document are the mnemonics corresponding. The operand stack is associated with a Java virtual machine.
3. Java bytecode instructions (hereinafter referred to as bytecode instructions): A bytecode instruction consists of an opcode and zero or more operands. Some bytecode instructions have no operands.
4. Java bytecode instruction sequence (hereinafter referred to as bytecode instruction sequence): It is a sequence composed of bytecode instructions, which are transformed from source code and corresponds to the function of source code one by one.
5. Opcode sequence: The operands of a bytecode instruction sequence are removed, and the rest is the opcode sequence.

In the next section, we will explore the advantages of bytecode-based detection.

### 1.2. Bytecode Features Analysis

Since source code is compiled without spaces, comments, and other formatting elements, T1 clone pairs can be identified by their corresponding opcode sequences in bytecode, making it easier to detect T1 clones.

According to Java virtual machine specification, most opcodes contain the type information of the data they operate on. For example, "istore" means to store int data from the operand stack to the local variable table, and "fstore" is used to operate float data. It can be

found that operations of “fstore” and “istore” have the same meaning, but the data types of operations are different. Most opcodes related to data types use special characters to indicate which type of data the opcode serves: “i” for int data, “l” for long data, “s” for short data, “b” for byte data, “c” for char data, “f” for float data, “d” for double data and “a” for reference type data [10].

Figure 1 shows opcode sequences corresponding to different types of “add” functions. The left parts of (a) and (b) are source code, and the right parts are corresponding opcode sequences. The added variables are int type in (a) and are double type in (b).

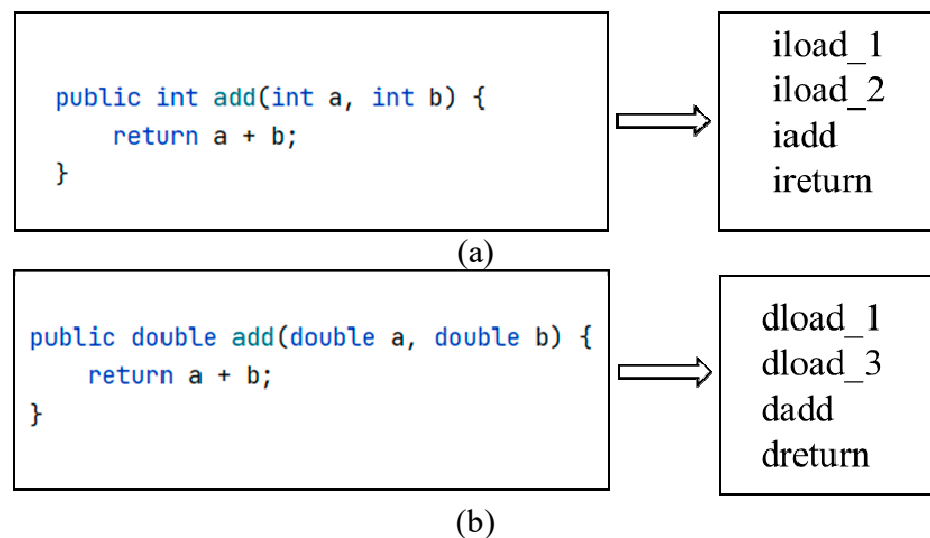


**Figure 1.** Example of source codes and opcodes in different types of “add” function: (a) The parameter type is “int”; (b) The parameter type is “double”.

These opcode sequences can be abstracted as:

$$\text{Sequence} = \{\text{Tload\_} < n >, \text{Tload\_} < n >, \text{Tadd}, \text{Treturn}\} \quad (1)$$

“T” is the abstraction of the data type operated, and “<n>” is the operand embedded in the opcode. The first two “Tload\_<n>” in sequence means to push two local variables of type T to the top of the stack. “Tadd” means to add two variables of type T and sends the result to the top of the stack. “Treturn” means the end of the function and returns the top value of the stack. It can be seen from Equation (1) that “add” functions of different types essentially correspond to the same abstract opcode sequence. If the semantic similarity between opcodes can be used, it will be easier to detect such clones, and T2 clones belong to this category. In addition, from Equation (1), it can be found that the opcode sequence is in order. It is necessary to load variables into the operand stack first, then perform an arithmetic operation, and finally return the operation result. Recurrent neural networks (RNNs) are good at extracting information from sequences, which means that perhaps RNNs can be used to extract features from opcode sequences.

Source code is a high-level program language for humans, and bytecode is a binary code for Java virtual machine. However, both of them have their own advantages. Source code is easy to understand, friendly to humans, and has strong logical expression ability. Bytecode eliminates the syntactic sugar of source code, which is concise and clear.

Figure 2 shows two different implementations of exponentiation functions, which correspond to the same opcode sequence. Figure 2a implements exponentiation with a “for” loop, and Figure 2b implements exponentiation with a “while” loop. Although at the source code level, exponentiation functions have different representations, but the corresponding bytecodes are exactly the same. There are many similar situations. For example, “++” and “+=” correspond to the same opcode “iinc”, and it can be found that program structures with the same semantics correspond to the same bytecodes.

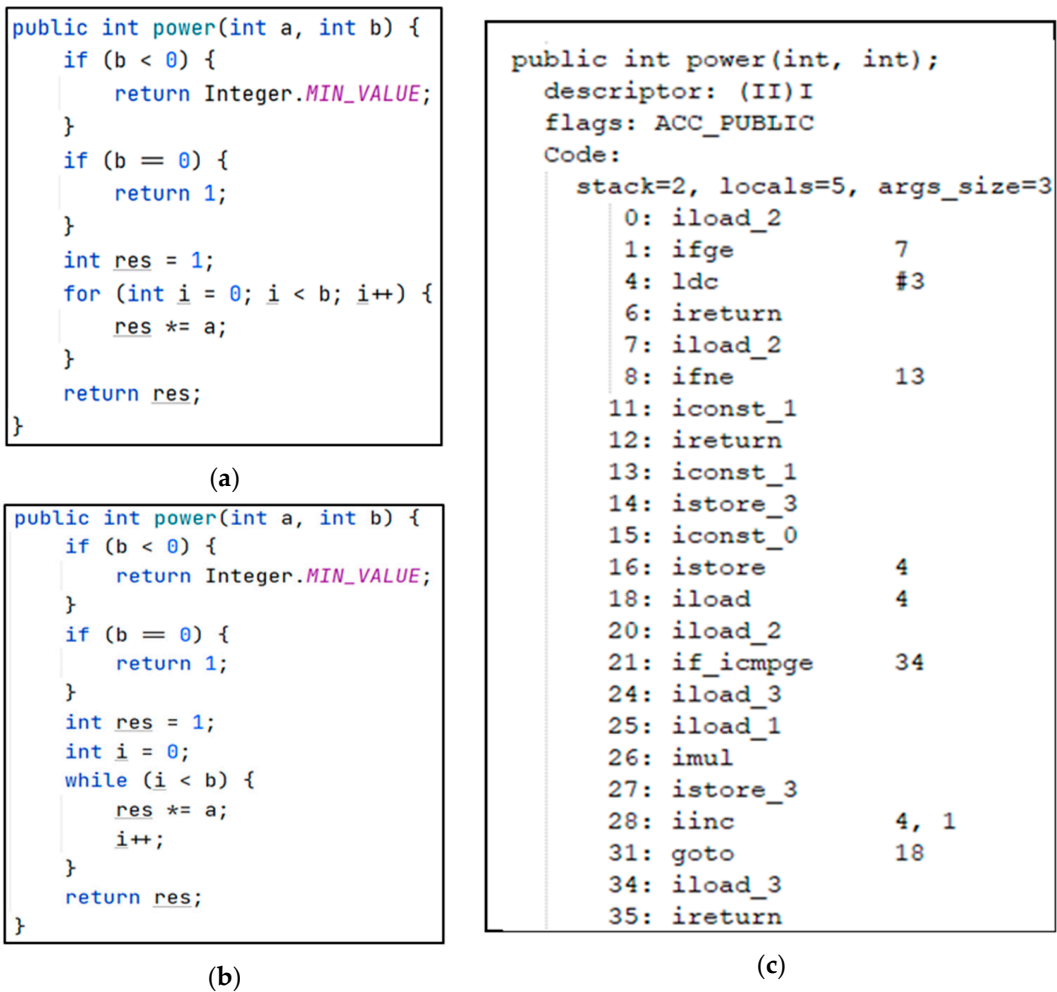


**Figure 2.** Correspondence between source code and bytecode instruction of exponentiation function isomorphism. Different exponentiation functions correspond to the same bytecode: (a) Function 1; (b) Function 2; (c) The bytecode corresponding to functions 1 and 2.

In addition to semantically identical program structures, syntactic sugar also enables different source codes to correspond to identical bytecode. There are a lot of syntactic sugars in Java source code, which use concise statements to express complex meanings. For example, the splicing function of String type can be implemented using “+”, which converts the String type to the StringBuilder type. StringBuilder uses “append ()” function to splice character string, and finally calls the “toString ()” function to convert the splicing result to String type. When source code is compiled, the syntactic sugar will be restored to the original operation, that is, the bytecode will show the essence of the syntactic sugar.

Program structures with the same semantics makes the implementation of programs more diverse. Syntactic sugar can help programmers develop quickly. However, these bring certain obstacles to code clone detection at the source code level. Program structures and syntactic sugars with the same semantics correspond to the same bytecode, so bytecode has advantages in semantic clone detection. Some Type-3 clones and all Type-4 clones belong to this category.

In summary, the following conclusions and inferences can be made:

1. Type-1 clones are detected easily at the bytecode level.
2. Type-2 clones are easy to detect if we use the semantic information of opcodes.
3. Type-3 and Type-4 clones can be detected by using bytecode.
4. Opcode sequences are ordered, so RNNs can be used to extract information from them.

In order to perform code clone detection in the case of missing source code and ensure the effect of code clone detection, we proposed a code clone detection method for Java bytecode (hereinafter referred to as bytecode): Siamese Neural Network [11] for Java Bytecode Clone Detection (SJBCD). SJBCD trains an opcode word vector model and constructs a GRU-based Siamese neural network for code clone detection. In order to verify the detection effect of the detection model constructed, we evaluate SJBCD using the public dataset BigCloneBench [12] and proves the effectiveness of our method. Related programs and data have been open-sourced to Github (Refer to the Data Availability Statement section).

## 2. Related Works

According to different code representation methods, code clone detection methods can be divided into methods based on text, token, abstract syntax trees, and graph (program flow graphs, program dependency graphs, etc.).

The main idea of text-based methods is to treat source code as a sequence of characters and use the difference between two character sequences as the difference between code fragments. Representative methods include NICAD [13], Duploc [14], SSD [15] and so on. After normalizing the code, NICAD uses the Longest Common Subsequence algorithm to compare the text lines of potential clones. Duploc first removes comments, whitespace, etc., and then uses a string-matching algorithm to detect code clones. The above two methods have good detection effects for T1 and T2 clones, but the results are not ideal when detecting code clones at the syntax and semantic levels.

The main idea of token-based methods is to extract the information from source code and present it in the form of word sequences. Common token-based detection methods include CCAaligner [16], CCLearner [17], and so on. CCLearner customizes token rules, uses reserved words, method identifiers, variable identifiers, and some abstract syntax trees information as features, performs numerical calculations in groups, and puts them into a deep neural network model for supervised training. CCLearner has advantages over text-based detection methods. In recent years, token methods are often combined with other program information and do not appear alone.

The detection methods based on tree and graph transform source code into abstract syntax trees, program flow graphs, program dependency graphs, etc. for clone detection. Recently, methods based on trees and graphs include TBCCD [18], Holmes [19], Raheja et al. [20], and so on. TBCCD converts source code into Abstract Syntax Trees (AST), which are then represented as word vectors and trained using a convolutional network. Holmes extracts program dependency graphs of source code through soot [21], and combines graph neural network technology and Siamese neural network to detect code clones. These methods have good detection effects on Type-3 and Type-4 clones.

In addition to the above methods detecting source code clones, there are also some methods based on compiled code. Zhang et al. [22] proposed a binary file code clone detection method based on the suffix tree for C language. The method disassembles binary executable files to obtain assembly instruction sequences, opcode sequences, and instruction type sequences and then constructs suffix trees of these sequences for code clone detection. In Java language, some scholars have also conducted research on code clone detection in binary code. Java language uses virtual machine technology, and bytecode is a special binary code for Java virtual machine. Yu et al. [9] conducted related research and proposed a code clone detection method based on bytecode sequence alignment, which uses the Smith-Waterman algorithm to align bytecode sequences for accurate matching, and the method also considers the similarities between instruction sequences and method calling sequences. However, the semantic information of opcodes is not considered. These methods can perform code clone detection in scenarios where only compiled code instead of source code can be obtained.

With the development of deep learning technologies, code clone detection methods are gradually approaching the direction of machine learning and deep learning from traditional

methods. Deep learning technologies are increasingly combined with code clone detection technology. Type-3 and Type-4 clones that were once difficult to detect can also be detected with higher precision and recall [23]. DeepSim [24] encodes the control flow and data flow of the code into a semantic matrix, where each element represents a high-dimensional sparse binary feature vector. Additionally, a novel deep-learning model is designed to quantify code similarity. SEED [25] addresses the specific characteristics of Type-4 clones by constructing a semantic map for each code fragment using an intermediate representation. This semantic map places emphasis on operators and API calls, rather than encompassing all tags. Subsequently, SEED utilizes a graph-deep neural network to generate feature vectors for the purpose of code clone detection. ASTNN [26] employs a technique wherein a large AST is divided into a sequence of smaller statement trees. These statement trees are then encoded into vectors, capturing the vocabulary and syntax knowledge of the individual statements. Next, a bidirectional RNN model utilizes the sequence of statement vectors to generate vector representations for code snippets. FA-AST [27] enhances the original AST by constructing a graph representation of the program. This approach effectively utilizes the structural information of the code snippet. Subsequently, a graph neural network is utilized to detect code similarity, leveraging the comprehensive structural features captured by the enhanced graph representation. Code-Token-Learner [28] introduced a code marker learner aimed at automatically reducing the number of feature markers used. They also proposed a tree-based position embedding method considering the tree-like structure of the abstract syntax tree. This method effectively encodes the position of each tag in the input. Moreover, in addition to converters that capture code dependencies, an essential component called the cross-code attention module is employed to capture similarities between two code snippets.

### 3. Methodology

By analyzing the sequence of bytecode instructions or opcodes, code clones can be detected and analyzed. A bytecode-based method for code clone detection focuses on similarity analysis of code logic and structure, while disregarding the specific details of the source code. The code clone detection process typically involves two common stages: code characterization and clone detection. SJBCD method follows a similar two-step approach. Firstly, it involves extracting the sequence of opcodes from the bytecode. Then, a Siamese neural network model is utilized to detect code clones. The advantage of using bytecode in code clone detection lies in its ability to capture the core logic and structure of the code, without being affected by variations in the source code representation. As a result, it allows for a more precise and efficient detection of code clones.

Code clone detection methods generally include two steps: the code characterization stage and the clone detection stage [29]. SJBCD also consisted of two steps: extracting opcode sequences and detecting code clones using the Siamese neural network model, as shown in Figure 3. SJBCD first obtained opcode sequences from bytecode files, then put opcode sequences into a Siamese neural network, and determined whether the function pair corresponding to the opcode sequence pair is a code clone according to the threshold set.

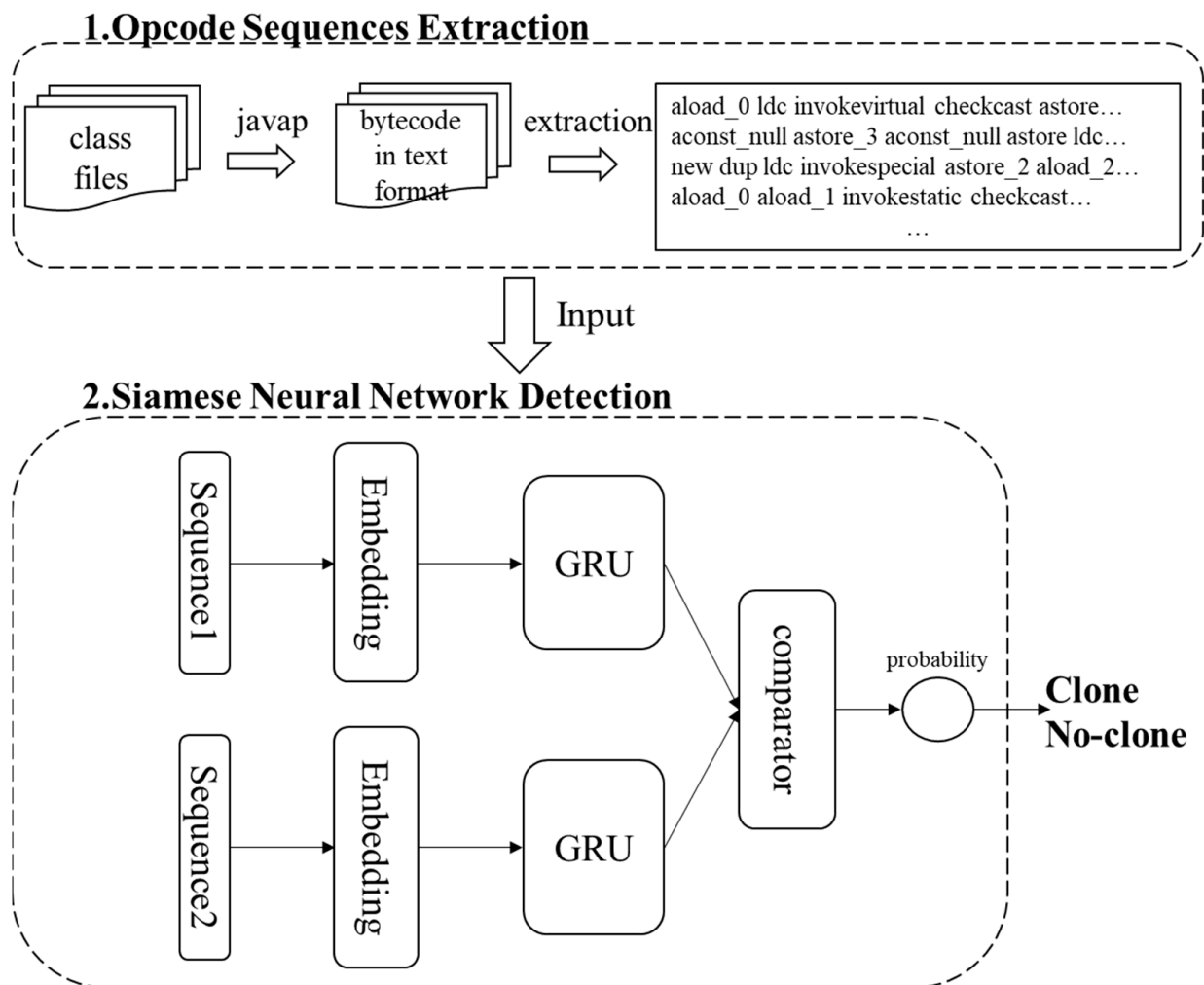


**Figure 3.** Two stages of SJBCD.

### 3.1. Extracting Opcode Sequences

SJBCD used opcode sequences to represent functions of source code. The opcode sequences extraction process is shown in Figure 3. Firstly, class binary files were converted into bytecode instruction files in text format through “javap” command. “Javap” is a tool carried by JDK, which has similar disassembly capabilities and can convert binary bytecode oriented to Java virtual machines into easy-to-understand assembly-like codes. Then opcode sequences were extracted from bytecode instruction in text format through regular matching. In order to facilitate the positioning of clone pairs and non-clone pairs, this paper also extracts the function-related information corresponding to the sequence when extracting the opcode sequence: modifier list, return value type, function name, parameter list, and file path where the function is located. The detailed extraction process of opcode sequences is shown in Algorithm 1.

**Algorithm 1** Opcode sequences extraction algorithm

---

```

Input a project that contains some class files.
repeat
    txt file  $\leftarrow$  exec "javap -verbose -p" class file
    Read txt file.
    repeat
        if line is function signature then
            sequence.signature  $\leftarrow$  line
            repeat
                opcodes add line.
            until line not match ("word number" or "word")
            sequence.opcodes  $\leftarrow$  opcodes
            sequences add sequence.
        end if
    until current line is last
    opcode sequences add sequences.
until current class file is last
Output opcode sequences.

```

---

We process an input item that includes a class file and iterate over each class file, converting them into text-form bytecode instruction files, specifically stored as .txt files. Since each opcode is represented by a single line in the file, we can traverse the content line by line. By separating them based on function signatures, we can collect all functions within each .txt file. The opcode always follows the format of "word number" or "word," which enables us to extract the opcode from each line. Essentially, combining these opcodes forms a sequence that represents a specific function. By organizing all the opcode sequences present in the project, we can successfully accomplish the initial step of the extraction process.

### 3.2. Building GRU-Based Siamese Neural Network

We referred to the open-source code idea of calculating Chinese text similarity on Github (Refer to the Data Availability Statement section) and built a detection model for bytecode clone detection. Our detection model consisted of two jobs: training an opcode word vector model and building a GRU-based Siamese Neural Network, which are described in detail below.

#### 3.2.1. Generating Opcode Word Vector Model

There is a semantic affinity between different opcodes. In order to utilize the semantic information of opcodes, GloVe [30] was used to represent opcodes in this paper. GloVe trains the word vector model based on the co-occurrence matrix, which has the characteristics that train fast and make full use of the relationship between words. Training GloVe needs a corpus. The quality of the corpus plays a crucial role as the foundation of the model, directly influencing the effectiveness of the detection model. In the case of Java language, the JDK serves as a reliable and natural source for building a corpus. For our research, we constructed the corpus using Oracle JDK 1.8. However, it is important to note that readers have the flexibility to select a different JDK version based on their specific code environment for testing purposes.

As shown in Figure 4, firstly, we obtained the class file library according to Oracle JDK 1.8 source code. We can also obtain class files from "rt.jar" file in JDK 1.8. Secondly, the corresponding textual opcode library was obtained by "javap" command. Finally, opcode sequences of all functions were extracted by regular matching. Opcodes were separated by spaces. The data volume of the corpus prepared for GloVe is 159,302. Oracle JDK 1.8 is an open-source Java development tool of Oracle Corporation.

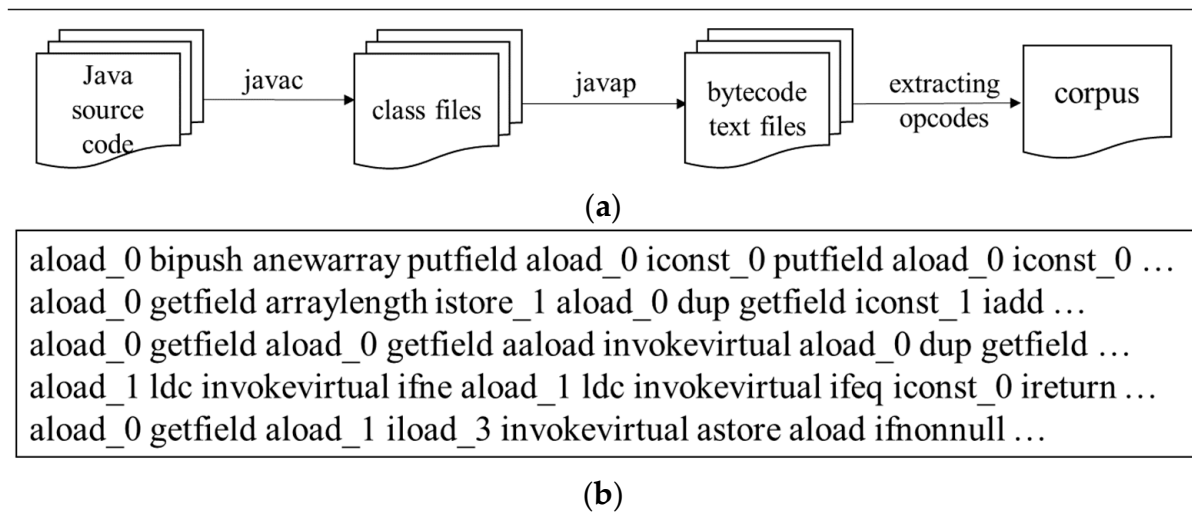


**Figure 4.** Extraction process of opcode word vector corpus: (a) Corpus extraction process; (b) Bytecode sequence results.

We input the corpus into the Glove system and train it to obtain a word vector model. This model essentially functions as a dictionary, capturing the relationship between each opcode and its corresponding word vector representation. In subsequent tasks, the detection process does not rely on the raw opcode text itself, but rather on the utilization of the associated word vectors.

### 3.2.2. Building Siamese Neural Network

Deep learning technology is increasingly used in the field of code clone detection, and the Siamese neural network technology [23] has been frequently used lately, achieving a good detection effect [18,26,31]. A Siamese neural network is a neural network structure that contains two identical subnetworks that share weights, which has the advantage of being able to handle pairs of input data and having fewer weight parameters. Gated Recurrent Units (GRU), as one of the representatives of RNNs, not only inherits the advantages of RNNs in processing sequence information but also adds the ability to learn long-term dependencies. Compared with a long short-term memory model (LSTM), GRU has fewer training parameters and is faster to train. We compared the two in our model and found that the detection effect is not much different, but GRU converges faster. Therefore, we built a Siamese neural network model based on GRU. The model structure is shown in Figure 5.

Firstly, we input paired opcode sequences to the model, and opcode sequences were converted into vector sequences of length  $200 \times 300$  by the word vector pretrained model. Secondly, the model fed the sequences of vectors into the GRU layer, which converted them into  $175 \times 1$  vectors. Finally, the model fed vectors into subsequent comparison networks. In the comparison network, a variety of distance calculation methods can be used, such as cosine distance, Euclidean distance, etc. We can also use the “concatenate” method commonly used in deep learning techniques to concatenate the paired vectors output by the GRU layer. We designed two model variants, one that spliced the output of the GRU module called SJBCD, and the other that calculated the cosine distance of the output called SJBCD-cos. The cosine distance is as follows:

$$\cos(\theta) = \frac{X_1 \cdot X_2}{\|X_1\| \|X_2\|} \quad (2)$$

$$\frac{X_1 \cdot X_2}{\|X_1\| \|X_2\|} = \frac{\sum_{i=1}^n X_1^i \cdot X_2^i}{\sqrt{\sum_{i=1}^n (X_1^i)^2} \sqrt{\sum_{i=1}^n (X_2^i)^2}} \quad (3)$$

where  $X_1$  and  $X_2$  are vectors output by GRU.  $X_1^i$  and  $X_2^i$  are the elements of vectors.

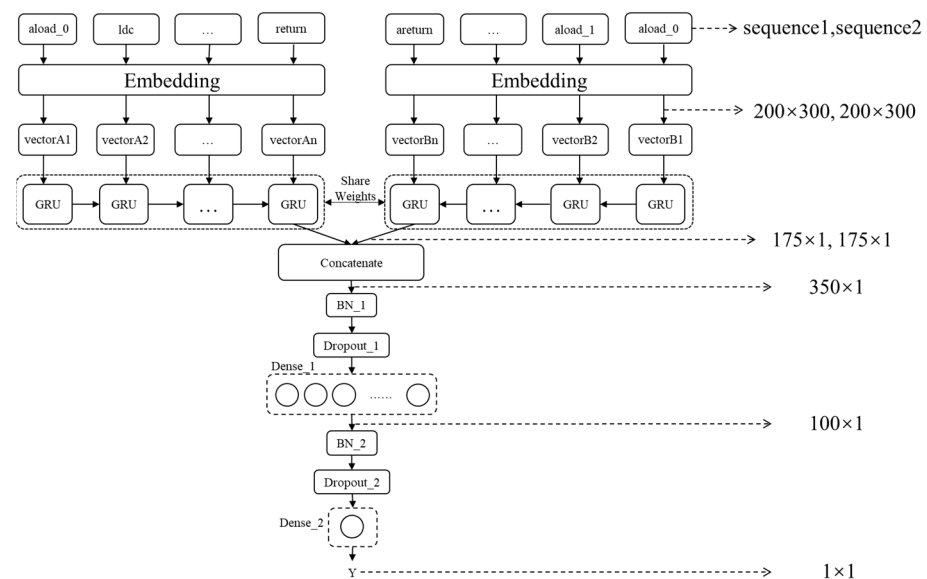


**Figure 5.** GRU-based Siamese neural network structure.

To improve training speed and prevent overfitting, we added the BN layer and Dropout layer to the comparison network. These techniques are common in deep learning and help achieve better model detection results.

We put the opcode sequence pair into the trained model, obtained a probability in the range of  $[0, 1]$ , and judged whether the opcode sequence pair was a clone pair based on a threshold.

Binary cross entropy is a loss function commonly in classification problems. Our model is essentially a binary classifier, so binary cross entropy is used as the loss function. The calculation formula is as follows:

$$\text{loss} = y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log(1 - \hat{y}_i) \quad (4)$$

where  $y_i$  is the actual value and  $\hat{y}_i$  is the predicted value.

In general, the SJBCD method can be divided into two steps. Firstly, the SJBCD method obtained opcode sequences of functions through multiple translation and text extraction methods and used word embedding technology to obtain the semantic information of opcodes. Secondly, a Siamese neural network based on GRU was used to predict whether opcode sequence pairs were clones.

#### 4. Experiment

For a comprehensive comparison, we conducted evaluations of our methods on compiled datasets as well as a common dataset. In this section, we will provide a detailed overview of the experimental setup. Firstly, we present the composition of the compiled dataset, CompiledBCB, that was used in our experiments. Next, we discuss the performance of the SJBCD method on three different word vectors. Finally, we compare the effectiveness of the SJBCD method with other existing methods.

##### 4.1. Datasets

BigCloneBench is a dataset constructed by Svajlenko et al. They mined the clone code of specific functions and marked the clone tag of code fragment pairs manually. It is still expanding on Github (Refer to the Data Availability Statement section) and is a benchmark dataset [12]. Recently, most methods based on source code have been evaluated by this

dataset. However, this dataset cannot be compiled, that is, it is impossible to evaluate clone detection methods based on bytecode.

JCoffee is a Java code repair tool proposed by Piyush et al. It tries to convert code fragments into compilable programs and works with any well-typed code fragments (class, function, or even an unenclosed group of statements) while making minimal changes to the input code fragment. JCoffee leverages compiler feedback to convert partial Java programs into their compilable counterparts by simulating the presence of missing surrounding code [32]. Figure 6 shows how using JCoffee fixes the code.

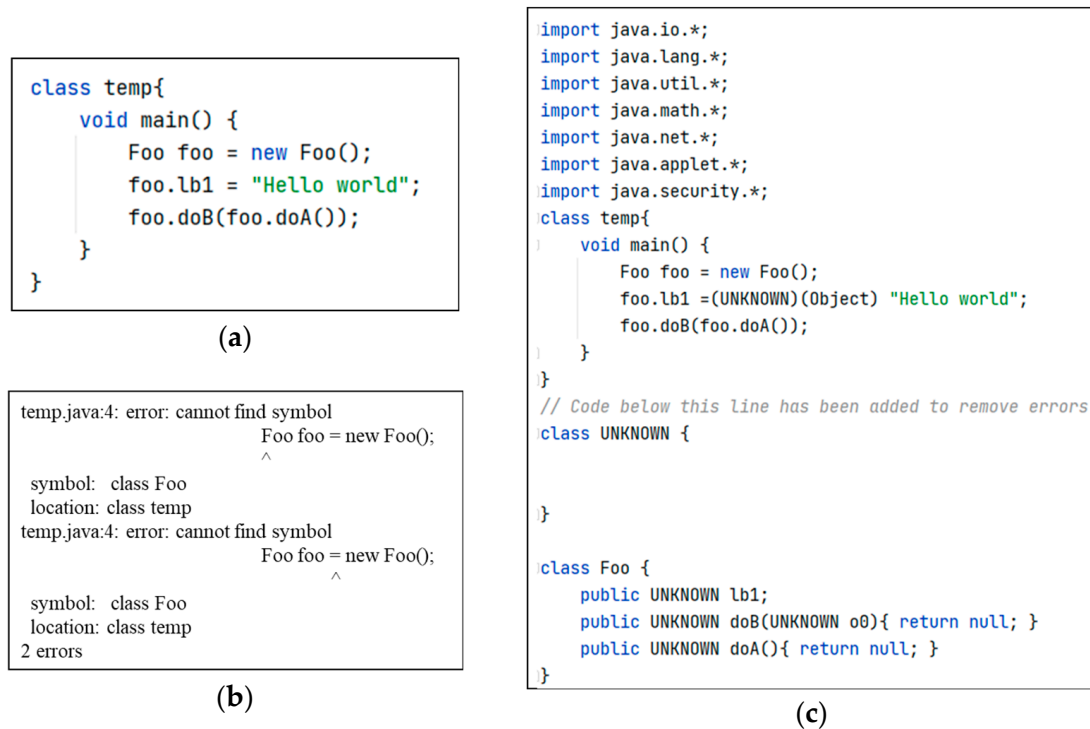


**Figure 6.** JCoffee code fix example: (a) Snippet of code to be repaired; (b) Information about initial compilation; (c) Fix the code snippet after completion.

Figure 6a represents the original code that needs fixing, Figure 6b displays the error message generated by the Java compiler during the initial compilation attempt, and Figure 6c showcases the repaired code. It can be observed that JCoffee primarily performs two tasks for code repair. Firstly, it imports the necessary Java dependency packages to resolve compilation errors. Secondly, it creates the missing classes based on the compiler prompts, ensuring that the original code can be successfully compiled. From Figure 6b, it can be observed that during the initial compilation, the compilation environment indicates that the class “Foo” is missing. Therefore, the JCoffee tool adds the “Foo” class and performs another compilation. Subsequent compilations may still produce relevant prompts, and thus, by continuously compiling and addressing these prompts, the missing code information can gradually be completed.

Since the main comparison is made between our model and TBCCD, in order to make the experimental environment as same as possible, we used JCoffee to try to compile the dataset it used. TBCCD extracted part of the data of BigCloneBench for model training and testing, of which 98.23% were Weak Type-3/Type-4 clones, and the rest were other types of clones. The distribution of clone types in this dataset is shown in Table 1 [18].

**Table 1.** Proportion of each clone type in the dataset used by TBCCD.

| Clone Type  | T1    | T2    | VST3  | ST3  | MT3   | WT3/T4 |
|-------------|-------|-------|-------|------|-------|--------|
| Percent (%) | 0.455 | 0.058 | 0.053 | 0.19 | 1.014 | 98.23  |

The clone pairs and non-clone pairs in the dataset consist of 9134 code fragments with a granularity of functions. There is an important parameter “-n” in JCoffee, which represents the number of attempts to compile during the repair process. We used JCoffee to write an automated program to perform repairs for three times. The parameters were 10, 20, and 40, respectively, and the code fragments for the three successful repairs were all 3878. To facilitate the experiment, we removed the code containing an anonymous inner class. There were still 3824 codes left. Each piece of data in the dataset consisted of two code fragments and a clone tag. According to these 3824 codes and the data records in the original data set, 423,217 pieces of data were obtained, which was called CompiledBCB. There were 2,416,589 pieces of data used in the original paper of TBCCD. According to Table 1, T1, T2, VST3, ST3 and MT3 accounted for 1.77%, and the total number was about 42,774. Assuming that the 42774 pieces of non-WT3/T4 data were kept in CompiledBCB, then the Weak Type-3/Type-4 clone data in CompiledBCB was 380,443 pieces, accounting for about 89.9%. The actual Weak Type-3/Type-4 clone data accounted for more than 89.9%.

#### 4.2. Word Vector Model Experiment

In order to select the most suitable word vector model, a word vector comparison experiment was conducted using the CompiledBCB dataset. Various dimensions of the GloVe, Skip-gram [33], and CBOW [33] models were combined with Siamese neural networks. The evaluation was based on accuracy, recall, and F1 measurements as indicators. The CompiledBCB dataset contained a vast amount of data, which led to slow training times. To expedite the selection process for suitable word vector dimensions and models, we employed the Inject Mutatio [34] method using JDK 1.8 as input. This approach facilitated the generation of Opcode21K, a smaller dataset consisting of 21K instances. With its reduced size, the Opcode21K dataset enabled faster training and evaluation processes.

Since the F1 metric combines accuracy and recall, the F1 metric was prioritized in the evaluation. If the F1 metric was equal, the accuracy and recall were then compared. Table 2 presents the test results of the three models at different word vector dimensions.

**Table 2.** Test results of three models in different word vector dimensions.

| Model     | Dimensions | Accuracy | Recall | F1           |
|-----------|------------|----------|--------|--------------|
| GloVe     | 100        | 0.986    | 1      | 0.993        |
| GloVe     | 150        | 0.989    | 1      | 0.994        |
| GloVe     | 200        | 0.991    | 1      | <b>0.995</b> |
| GloVe     | 250        | 0.99     | 1      | <b>0.995</b> |
| GloVe     | 300        | 0.99     | 1      | <b>0.995</b> |
| Skip-gram | 100        | 0.983    | 1      | 0.992        |
| Skip-gram | 150        | 0.986    | 1      | 0.993        |
| Skip-gram | 200        | 0.99     | 1      | <b>0.995</b> |
| Skip-gram | 250        | 0.986    | 1      | 0.993        |
| Skip-gram | 300        | 0.986    | 1      | 0.993        |
| CBOW      | 100        | 0.981    | 1      | 0.99         |
| CBOW      | 150        | 0.985    | 1      | 0.992        |
| CBOW      | 200        | 0.989    | 1      | 0.994        |
| CBOW      | 250        | 0.985    | 1      | 0.992        |
| CBOW      | 300        | 0.987    | 1      | 0.993        |

To visually illustrate the changing trend of the detection effectiveness for the three models with word vector dimension, this study extracted the F1 metric values from Table 2 and generated Figure 7.

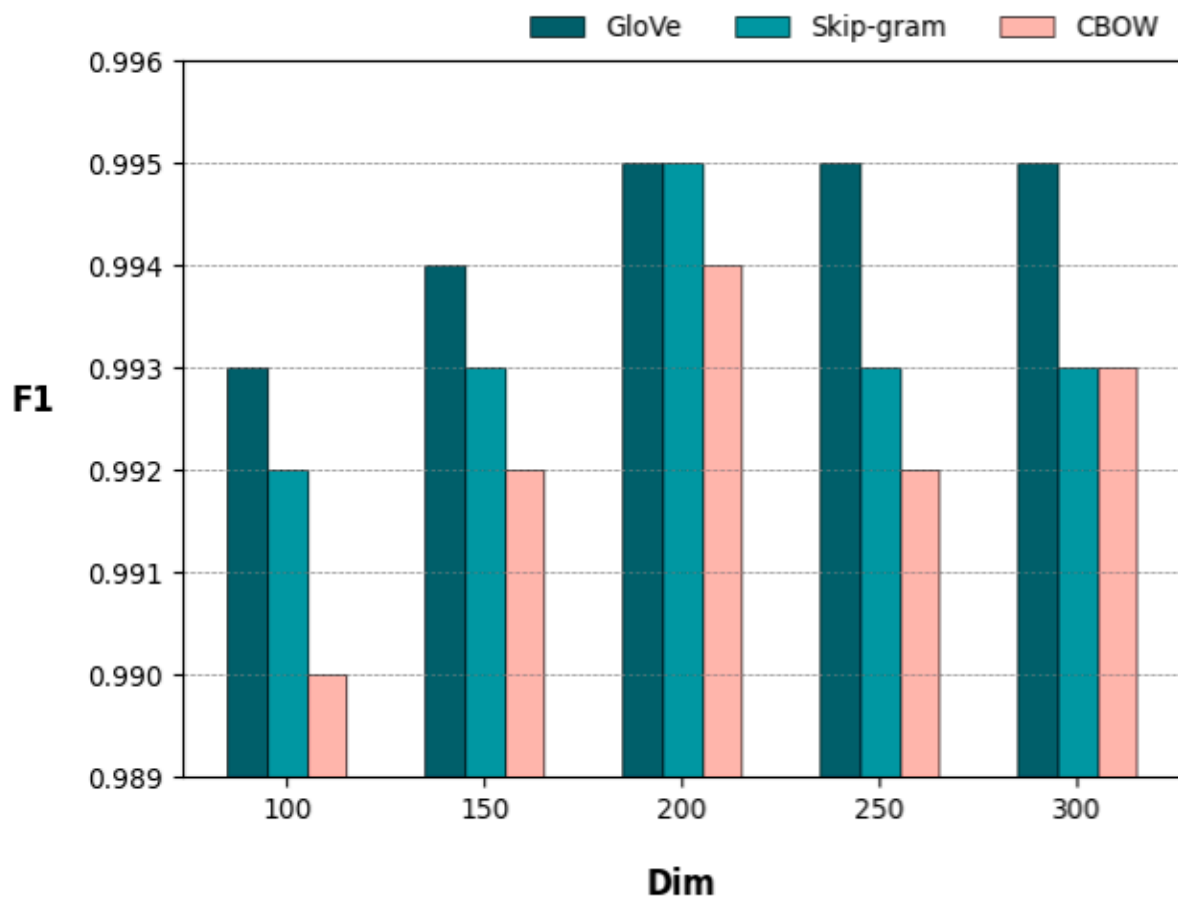


**Figure 7.** The variation trend of the F1 metric value of three models under different word vector dimensions.

It is evident from Figure 7 that both Skip-gram and CBOW models achieve their peak F1 metric values at a dimension of 200 VDS, which are 0.995 and 0.994, respectively. In contrast, the GloVe model reaches its highest F1 metric value at 200 VDS and remains stable at 0.995 for 250 and 300 dimensions. Table 2 indicates that the accuracy of the 200-dimension word vector is the highest at 0.991. Hence, it can be concluded that 200 dimensions are the optimal choice for the word vector dimension.

Furthermore, Figure 7 reveals that when the word vector dimension is set to 200, the F1 metric values of the GloVe and Skip-gram models are equal. Further examination of Table 2 indicates that the accuracy of the GloVe model at this point is 0.991, which is higher than the accuracy of Skip-gram (0.99). Consequently, GloVe is selected as the pre-training model for word vectors in this study.

#### 4.3. Methods Experiment and Results

According to the evaluation results of the literature [6], it can be found that compared with other clone detection tools and methods, the detection effect of NICAD on T1-3 clones is at the forefront, and NICAD is still updated on Github continuously. This paper selected the latest version 6.2 for experiments. In addition, TBCCD and its variants were also selected for comparative experiments. TBCCD combines abstract syntax trees and a convolutional network to detect code clones. Compared with the previous method CDLH based on abstract syntax trees and deep learning, the F1-score is improved by 0.15 [18]. The performance is very good, and currently, only the Holmes method is slightly better, the F1-score of TBCCD is 0.95, and the Holmes method is 0.99. Holmes combines the program dependency graphs with the graph neural network fused with the attention mechanism to detect code clones. We attempted to reproduce Holmes, but the program

shared by the author on Google Drive lacks the conversion of source code to graphs. So, we have chosen TBCCD to reproduce. We also attempted to reproduce the detection method based on bytecode proposed by Yu et al. [9], but the code and data are not disclosed in their paper. CompileBCB dataset contains both source code data and bytecode data, so code clone detection methods based on bytecode can be compared with those based on source code.

In this paper, SJBCD was trained and tested on bytecode data, TBCCD was trained and tested on corresponding source code data, and NICAD was tested on source code data.

In addition to the NICAD and TBCCD methods, we also include other well-known methods for comparison, such as FA-AST [27], and the Code-Token-Learner [28], all of which have demonstrated excellent performance on the CompiledBCB dataset.

Table 3 shows the result of the comparative experiments.

**Table 3.** Experimental results of each detection method on CompiledBCB.

| Method             | Precision    | Recall       | F1-Score     |
|--------------------|--------------|--------------|--------------|
| SJBCD (ours)       | 0.991        | <b>0.997</b> | <b>0.994</b> |
| SJBCD-cos (ours)   | <b>0.993</b> | 0.995        | <b>0.994</b> |
| TBCCD              | 0.9          | 0.915        | 0.908        |
| TBCCD+token        | 0.98         | 0.953        | 0.966        |
| TBCCD+token-type   | 0.976        | 0.964        | 0.97         |
| TBCCD+token+PACE   | 0.971        | 0.957        | 0.964        |
| NICAD              | 0.636        | 0.005        | 0.01         |
| Code-Token-Learner | 0.984        | 0.933        | 0.958        |
| FA-AST             | 0.988        | 0.988        | 0.988        |

From the experimental results, it can be found that:

1. The detection effect of NICAD is much weaker than that of SJBCD and TBCCD, and its F1-score is only 0.01.
2. The F1-score of SJBCD is 0.994, which is 0.006 higher than FA-AST, which is 0.988.

To demonstrate the effectiveness of CompiledBCB, Table 4 [18] shows the experimental results of TBCCD on the original BigCloneBench dataset.

**Table 4.** The detection effect of TBCCD on BigCloneBench.

| Method               | Precision | Recall | F1-Score |
|----------------------|-----------|--------|----------|
| TBCCD                | 0.78      | 0.73   | 0.76     |
| TBCCD + token        | 0.95      | 0.95   | 0.95     |
| TBCCD + token-type   | 0.94      | 0.95   | 0.95     |
| TBCCD + token + PACE | 0.94      | 0.96   | 0.95     |

It can be known from Tables 3 and 4 that F1-score of TBCCD on CompiledBCB is 0.908, and F1-score on BigCloneBench is 0.76, differing by 0.148; F1-score of TBCCD+token on CompiledBCB is 0.966, and F1-score on BigCloneBench is 0.95, differing by 0.016; F1-score of TBCCD+token-type on CompiledBCB is 0.970, and F1-score on BigCloneBench is 0.95, differing by 0.02; TBCCD+token+PACE on CompiledBCB is 0.964, and F1-score on BigCloneBench is 0.95, differing by 0.014. It can be found that except for the slight difference in the TBCCD method, the difference in detection effects of other variants is very small on the two data sets, which indicates that CompiledBCB is effective.

Weak Type-3/Type-4 clones have less text similarity, so the text-based method NICAD does not work well on CompiledBCB. TBCCD extracts AST of source code, and our method utilizes opcode sequences. Both of these combine deep learning techniques. From the experimental results, they all have great detection effects. As shown in Table 3, the F1-score of SJBCD is 0.984 higher than that of NICAD, and 0.006 higher than that of FA-AST. In this

paper, the detection effect of the SJBCD method on Weak Type-3/Type-4 clones is better than that of existing detection methods, which is in line with expectations.

## 5. Conclusions

This paper proposed a code clone detection method that combines the semantic information of bytecodes with a Siamese neural network to calculate the similarity between programs. Its characteristics are that it can detect the situation where source code is missing and ensure the detection effect on Weak Type-3/Type-4 clones. When evaluated on the CompiledBCB dataset, SJBCD achieved an impressive recall rate of 0.997. This performance exceeds that of current popular methods such as FA-AST's 0.988. SJBCD is a new technology for Jar format code cloning detection.

However, it is important to acknowledge the limitations of this paper. For instance, it focuses solely on detecting code clones in Java, thereby restricting its applicability to other programming languages. Additionally, the impact of code detection after obfuscation is not extensively discussed, indicating a potential area for further exploration.

In the future, we intend to delve into cross-language code clone detection, expanding the application of our method to encompass other programming languages and discussing the detection effect on obfuscated code. As a binary code for Java virtual machine, bytecode is not only used in Java programs but also in Scala, Groovy, Kotlin, and other languages. Therefore, SJBCD can be applied to program languages using JVM. In addition to this, code clone detection of cross-language between these program languages can also be performed using SJBCD. The compiled code of C, C++, and other languages can be processed into assembly code. In the future, SJBCD is expected to be extended to compiled program languages such as C language. Given that Java projects are frequently obfuscated, we will also focus on enhancing code clone detection specifically for obfuscated code.

**Author Contributions:** Conceptualization, B.W. and Y.Q.; methodology, B.W. and S.D.; formal analysis, S.D. and J.Z.; data curation, S.D. and J.Z.; software, S.D. and J.Z.; writing—original draft, S.D. and J.Z.; writing—reviewing and editing, B.W.; supervision, Y.Q. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the Chongqing Construction Science and technology plan project of Chongqing Housing and Urban-Rural Development Commission under Grant No. CKZ 2021 2-9.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** SJBCD method presented in this study can be reproduced with code openly available at <https://github.com/withsunny/SJBCD>. Reference of Chinese text similarity calculation of open source code can be found at <https://github.com/zqhZY/semanaly>. The dataset BigCloneBench can be found at <https://github.com/clonebench/BigCloneBench>.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Ain, Q.U.; Butt, W.H.; Anwar, M.W.; Azam, F.; Maqbool, B. A Systematic Review on Code Clone Detection. *IEEE Access* **2019**, *7*, 86121–86144. [\[CrossRef\]](#)
2. Chen, C.F.; Zain, A.M.; Zhou, K.Q. Definition, approaches, and analysis of code duplication detection (2006–2020): A critical review. *Neural Comput. Appl.* **2022**, *34*, 20507–20537. [\[CrossRef\]](#)
3. Dang, Y.; Zhang, D.; Ge, S.; Huang, R.; Chu, C.; Xie, T. Transferring Code-Clone Detection and Analysis to Practice. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), Buenos Aires, Argentina, 20–28 May 2017; pp. 53–62. [\[CrossRef\]](#)
4. Zhang, H.; Sakurai, K. A Survey of Software Clone Detection from Security Perspective. *IEEE Access* **2021**, *9*, 48157–48173. [\[CrossRef\]](#)
5. Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.* **2007**, *33*, 577–591. [\[CrossRef\]](#)

6. Svajlenko, J.; Roy, C.K. Evaluating clone detection tools with BigCloneBench. In Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, Germany, 29 September–1 October 2015; pp. 131–140. [\[CrossRef\]](#)
7. Keivanloo, I.; Roy, C.K.; Rilling, J. SeByte: A semantic clone detection tool for intermediate languages. In Proceedings of the 2012 20th IEEE International Conference on Program Comprehension (ICPC), Passau, Germany, 11–13 June 2012; pp. 247–249. [\[CrossRef\]](#)
8. Yu, D.; Wang, J.; Wu, Q.; Yang, J.; Wang, J.; Yang, W.; Yan, W. Detecting Java Code Clones with Multi-granularities Based on Bytecode. In Proceedings of the 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), 4–8 July 2017; pp. 317–326. [\[CrossRef\]](#)
9. Yu, D.; Yang, J.; Chen, X.; Chen, J. Detecting Java Code Clones Based on Bytecode Sequence Alignment. *IEEE Access* **2019**, *7*, 22421–22433. [\[CrossRef\]](#)
10. Lindholm, T.; Yellin, F.; Bracha, G. *The Java Virtual Machine Specification*; Java SE 8 ed.; Pearson Education Inc.: New York, NY, USA, 2014.
11. Bromley, J.; Guyon, I.; Lecun, Y.; Sackinger, E.; Shah, R. Signature verification using a “siamese” time delay neural network. *Adv. Neural Inf. Process. Syst.* **1994**, *7*, 737–744. [\[CrossRef\]](#)
12. Svajlenko, J.; Islam, J.F.; Keivanloo, I.; Roy, C.K.; Mia, M.M. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 28 September–3 October 2014; pp. 476–480. [\[CrossRef\]](#)
13. Roy, C.K.; Cordy, J.R. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In Proceedings of the 2008 16th IEEE International Conference on Program Comprehension, Amsterdam, The Netherlands, 10–13 June 2008; pp. 172–181. [\[CrossRef\]](#)
14. Ducasse, S.; Rieger, M.; Demeyer, S. A language independent approach for detecting duplicated code. In Proceedings of the IEEE International Conference on Software Maintenance—1999 (ICSM'99), Oxford, UK, 30 August–3 September 1999; pp. 109–118. [\[CrossRef\]](#)
15. Lee, S.; Jeong, I. SDD: High performance code clone detection system for large scale source code. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05), New York, NY, USA, 16–20 October 2005; pp. 140–141. [\[CrossRef\]](#)
16. Wang, P.; Svajlenko, J.; Wu, Y.; Xu, Y.; Roy, C.K. CCAliigner: A Token Based Large-Gap Clone Detector. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 1066–1077. [\[CrossRef\]](#)
17. Li, L.; Feng, H.; Zhuang, W.; Meng, N.; Ryder, B. CCLearner: A Deep Learning-Based Clone Detection Approach. In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17–22 September 2017; pp. 249–260. [\[CrossRef\]](#)
18. Yu, H.; Lam, W.; Chen, L.; Li, G.; Xie, T.; Wang, Q. Neural Detection of Semantic Code Clones Via Tree-Based Convolution. In Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC), Montreal, QC, Canada, 25–26 May 2019; pp. 70–80. [\[CrossRef\]](#)
19. Mehrotra, N.; Agarwal, N.; Gupta, P.; Anand, S.; Lo, D.; Purandare, R. Modeling functional similarity in source code with graph-based Siamese networks. *IEEE Trans. Softw. Eng.* **2022**, *48*, 3771–3789. [\[CrossRef\]](#)
20. Raheja, K.; Rajkumar, T. An Emerging Approach towards Code Clone Detection: Metric Based Approach on Byte Code. *Int. J. Adv. Res. Comput. Sci. Softw. Eng.* **2013**, *3*.
21. Lam, P.; Bodden, E.; Lhoták, O.; Hendren, L.J. The Soot framework for Java program analysis: A retrospective. In Proceedings of the Cetus Users and Compiler Infrastructure Workshop (CETUS 2011), Galveston Island, TX, USA, 10 October 2011.
22. Zhang, L.H.; Gui, S.L.; Mu, F.J.; Wang, S. Clone Detection Algorithm for Binary Executable Code with Suffix Tree. *Comput. Sci.* **2019**, *46*, 141–147. [\[CrossRef\]](#)
23. Le, Q.Y.; Liu, J.X.; Sun, X.P.; Zhang, X.P. Survey of Research Progress of Code Clone Detection. *Comput. Sci.* **2021**, *48*, 509–522. [\[CrossRef\]](#)
24. Zhao, G.; Huang, J. DeepSim: Deep learning code functional similarity. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018), New York, NY, USA, 4–9 November 2018; pp. 141–151. [\[CrossRef\]](#)
25. Xue, Z.P.; Jiang, Z.J.; Huang, C.L.; Xu, R.L.; Huang, X.B.; Hu, L.M. SEED: Semantic Graph Based Deep Detection for Type-4 Clone. In Proceedings of the Reuse and Software Quality: 20th International Conference on Software and Systems Reuse (ICSR 2022), Montpellier, France, 15–17 June 2022; pp. 120–137. [\[CrossRef\]](#)
26. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 783–794. [\[CrossRef\]](#)
27. Wang, W.; Li, G.; Ma, B.; Xia, X.; Jin, Z. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020; pp. 261–271. [\[CrossRef\]](#)

28. Zhang, A.; Fang, L.; Ge, C.; Li, P.; Liu, Z. Efficient transformer with code token learner for code clone detection. *J. Syst. Softw.* **2023**, *197*, 111557. [[CrossRef](#)]
29. Geoff, W. *Plague: Plagiarism Detection Using Program Structure*; School of Electrical Engineering and Computer Science, University of New South Wales: Kensington, NSW, Australia, 1988.
30. Pennington, J.; Socher, R.; Manning, C. GloVe: Global Vectors for Word Representation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; pp. 1532–1543. [[CrossRef](#)]
31. Wu, Y.; Zou, D.; Dou, S.; Yang, S.; Yang, W.; Cheng, F.; Liang, H.; Jin, H. SCDetector: Software functional clone detection based on semantic tokens analysis. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 21–25 December 2020; pp. 821–833. [[CrossRef](#)]
32. Gupta, P.; Mehrotra, N.; Purandare, R. JCoffee: Using Compiler Feedback to Make Partial Code Snippets Compilable. In Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), Adelaide, SA, Australia, 27 September–3 October 2020; pp. 810–813. [[CrossRef](#)]
33. Mikolov, T.; Chen, K.; Corrado, G.S.; Dean, J. Efficient Estimation of Word Representations in Vector Space. In Proceedings of the International Conference on Learning Representations, Scottsdale, AZ, USA, 2–4 May 2013.
34. Svajlenko, J.; Roy, C.K. The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis. *IEEE Trans. Softw. Eng.* **2021**, *47*, 1060–1087. [[CrossRef](#)]

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.